**DAY-4-PROGRAMS**

**1.Counting Elements**

**PROGRAM:**

def count\_elements(arr):

element\_set = set(arr)

count = 0

for x in arr:

if x + 1 in element\_set:

count += 1

return count

print(count\_elements([1, 2, 3]))

print(count\_elements([1, 1, 3, 3, 5, 5, 7, 7]))

**2.Perform String Shifts**

**PROGRAM:**

def string\_shift(s, shift):

net\_shift = 0

for direction, amount in shift:

if direction == 0:

net\_shift -= amount

else:

net\_shift += amount

net\_shift %= len(s)

return s[-net\_shift:] + s[:-net\_shift] if net\_shift != 0 else s

print(string\_shift("abc", [[0, 1], [1, 2]])) # Output: "cab"

print(string\_shift("abcdefg", [[1, 1], [1, 1], [0, 2], [1, 3]])) # Output: "efgabcd"

**3. Leftmost Column with at Least a One**

**PROGRAM:**

class BinaryMatrix:

def \_\_init\_\_(self, mat):

self.mat = mat

def get(self, row, col):

return self.mat[row][col]

def dimensions(self):

return [len(self.mat), len(self.mat[0])]

def leftmost\_column\_with\_one(binaryMatrix):

rows, cols = binaryMatrix.dimensions()

current\_row, current\_col = 0, cols - 1

leftmost = -1

while current\_row < rows and current\_col >= 0:

if binaryMatrix.get(current\_row, current\_col) == 1:

leftmost = current\_col

current\_col -= 1

else:

current\_row += 1

return leftmost

print(leftmost\_column\_with\_one(BinaryMatrix([[0, 0], [1, 1]]))) # Output: 0

print(leftmost\_column\_with\_one(BinaryMatrix([[0, 0], [0, 1]]))) # Output: 1

print(leftmost\_column\_with\_one(BinaryMatrix([[0, 0], [0, 0]]))) # Output: -1

**4. First Unique Number**

**PROGRAM:**

from collections import deque, Counter

class FirstUnique:

def \_\_init\_\_(self, nums):

self.queue = deque(nums)

self.count = Counter(nums)

def showFirstUnique(self):

while self.queue and self.count[self.queue[0]] > 1:

self.queue.popleft()

return self.queue[0] if self.queue else -1

def add(self, value):

self.queue.append(value)

self.count[value] += 1

firstUnique = FirstUnique([2, 3, 5])

print(firstUnique.showFirstUnique()) # Output: 2

firstUnique.add(5)

print(firstUnique.showFirstUnique()) # Output: 2

firstUnique.add(2)

print(firstUnique.showFirstUnique()) # Output: 3

firstUnique.add(3)

print(firstUnique.showFirstUnique()) # Output: -1

**5.Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree**

**PROGRAM:**

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def is\_valid\_sequence(root, arr):

def dfs(node, arr, index):

if not node or index >= len(arr) or node.val != arr[index]:

return False

if not node.left and not node.right and index == len(arr) - 1:

return True

return dfs(node.left, arr, index + 1) or dfs(node.right, arr, index + 1)

return dfs(root, arr, 0)

root = TreeNode(0, TreeNode(1, TreeNode(0, None, TreeNode(1)), TreeNode(1, TreeNode(0))), TreeNode(0, TreeNode(0)))

print(is\_valid\_sequence(root, [0, 1, 0, 1])) # Output: True

print(is\_valid\_sequence(root, [0, 0, 1])) # Output: False

print(is\_valid\_sequence(root, [0, 1, 1])) # Output: False

**6.Kids With the Greatest Number of Candies**

**PROGRAM:**

def kids\_with\_candies(candies, extraCandies):

max\_candies = max(candies)

return [candy + extraCandies >= max\_candies for candy in candies]

print(kids\_with\_candies([2, 3, 5, 1, 3], 3)) # Output: [True, True, True, False, True]

print(kids\_with\_candies([4, 2, 1, 1, 2], 1)) # Output: [True, False, False, False, False]

print(kids\_with\_candies([12, 1, 12], 10)) # Output: [True, False, True]

**7.Max Difference You Can Get From Changing an Integer**

**PROGRAM:**

def max\_diff(num):

s = str(num)

a = s

b = s

for digit in s:

if digit != '9':

a = s.replace(digit, '9')

break

for digit in s:

if digit != '1':

b = s.replace(digit, '0' if digit == s[0] else '1')

break

return int(a) - int(b)

# Test cases

print(max\_diff(555)) # Output: 888

print(max\_diff(9)) # Output: 8

**8.Check If a String Can Break Another String**

**PROGRAM:**

def check\_if\_can\_break(s1, s2):

s1, s2 = sorted(s1), sorted(s2)

return all(c1 >= c2 for c1, c2 in zip(s1, s2)) or all(c2 >= c1 for c1, c2 in zip(s1, s2))

# Test cases

print(check\_if\_can\_break("abc", "xya")) # Output: True

print(check\_if\_can\_break("abe", "acd")) # Output: False

print(check\_if\_can\_break("leetcodee", "interview")) # Output: True

**9.Number of Ways to Wear Different Hats to Each Other**

**PROGRAM :**

def number\_ways(hats):

from collections import defaultdict

dp = defaultdict(int)

dp[0] = 1

all\_mask = (1 << len(hats)) - 1

hat\_to\_people = defaultdict(list)

for i, hat\_list in enumerate(hats):

for hat in hat\_list:

hat\_to\_people[hat].append(i)

for hat in range(1, 41):

new\_dp = dp.copy()

for mask, ways in dp.items():

for person in hat\_to\_people[hat]:

if mask & (1 << person) == 0:

new\_dp[mask | (1 << person)] = (new\_dp[mask | (1 << person)] + ways) % MOD

dp = new\_dp

return dp[all\_mask]

# Test cases

print(number\_ways([[3, 4], [4, 5], [5]])) # Output: 1

print(number\_ways([[3, 5, 1], [3, 5]])) # Output: 4

print(number\_ways([[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]])) # Output: 24

**10.Construct Binary Search Tree from Preorder Traversal**

**PROGRAM:**

class TreeNode:

def \_init\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def bst\_from\_preorder(preorder):

def helper(lower=float('-inf'), upper=float('inf')):

nonlocal idx

if idx == len(preorder):

return None

val = preorder[idx]

if val < lower or val > upper:

return None

idx += 1

root = TreeNode(val)

root.left = helper(lower, val)

root.right = helper(val, upper)

return root

idx = 0

return helper()

# Function to print tree in inorder (used for testing)

def print\_inorder(node):

if node:

print\_inorder(node.left)

print(node.val, end=' ')

print\_inorder(node.right)

# Test cases

preorder1 = [8, 5, 1, 7, 10, 12]

preorder2 = [10, 5, 1, 7, 40, 50]

bst1 = bst\_from\_preorder(preorder1)

bst2 = bst\_from\_preorder(preorder2)

print("Inorder traversal of BST from preorder1:")

print\_inorder(bst1) # Output: 1 5 7 8 10 12

print("\nInorder traversal of BST from preorder2:")

print\_inorder(bst2) # Output: 1 5 7 10 40 50